

Projet : Gestion des budgets d'une ville

Nous imaginons un problème où l'équipe d'une mairie doit financer un ensemble de projets de manière efficace. Votre travail permettra de simuler le processus de proposition des différents projets, et de sélection des projets les plus efficaces.

Simulation d'une équipe municipale

Au sein de la mairie de Dauphine City, une petite équipe est en charge de proposer de nouveaux projets afin d'améliorer la vie des Dauphinois.e.s. Cette équipe se compose d'expert.e.s (qui peuvent proposer des projets dans leurs secteurs d'expertise), d'évaluateur.rice.s (qui évaluent les différents coûts des projets), et d'un.e élu.e qui estime le bénéfice du projet pour la municipalité.

Dans la suite, on va décrire différentes composantes de cette équipe et on va vous demander de proposer une implémentation. La description est suffisamment précise pour avoir une implémentation, mais suffisamment vague pour vous forcer à choisir le type d'objet adéquat (une classe, une classe abstraite, une interface), le type de relation entre ces objets, et la structure de ces objets.

Les projets. Un projet proposé pour la municipalité est décrit par plusieurs caractéristiques : un titre, une description, un secteur, un bénéfice (quantifié par une valeur numérique) attendu pour la collectivité, trois coûts associés : un coût économique, un coût social, et un coût environnemental. Les secteurs possibles auxquels un projet peut être rattaché (un seul secteur par projet) sont : *sport, santé, éducation, culture, et attractivité économique*.

L'équipe municipale. Toutes les personnes travaillant à la mairie possèdent un nom, un prénom, un âge, ... Nous distinguons les fonctions/métiers suivants :

- Les évaluateur.rice.s sont des personnes chargées d'évaluer un coût particulier d'un projet. Un.e évaluateur.rice est toujours spécialisé.e sur un type de coût (une seule spécialisation). Sa spécialisation peut donc l'amener à renseigner soit le coût économique d'un projet, soit le coût social d'un projet, ou enfin son coût environnemental.
- L'élu.e est une personne qui représente la municipalité et doit évaluer le bénéfice d'un projet pour la collectivité.
- Un.e expert.e est une personne rattachée à une liste de secteurs dans lesquels il/elle a des compétences et pour lesquels il/elle peut proposer des projets.

Une équipe municipale regroupe tous ces rôles : un.e élu.e, trois évaluateur.rice (un.e pour la question économique, un.e pour la question sociale, et un.e pour la question environnementale), et une liste d'expert.e.s. Ainsi une équipe possède toutes les compétences pour proposer et évaluer des projets dans différents secteurs.

L'équipe peut exécuter un cycle de simulation : les expert.e.s proposent des projets. Chaque évaluateur.rice attribue un coût au projet (coût économique, social, environnemental). L'élu.e attribue un bénéfice. Le projet est complet et peut être rajouté à la liste des projets étudiés par la municipalité.

Pour le projet. Réfléchissez aux différentes classes, interfaces, énumérations, etc, nécessaires pour représenter les différents concepts présentés plus haut. Dans un package `equipe`, coder ces différents éléments. Vous êtes libres de rajouter des classes, des méthodes, des attributs et champs statiques à ceux “suggérés” dans cette section. Vous respecterez les bonnes pratiques d’encapsulation. Vous devrez pouvoir générer un cycle de simulation en utilisant un processus stochastique, dont vous choisirez les paramètres.

Sac à dos multidimensionnel

La suite du projet se construit à partir d’un problème d’optimisation combinatoire, celui du **sac à dos multidimensionnel**.

Le problème du sac-à-dos est un problème classique d’optimisation combinatoire où l’on cherche à sélectionner un ensemble d’objets d’utilité maximale sous une contrainte de coût.

Formellement le problème est défini comme suit :

- *Input* : un budget B et un ensemble $O = \{o_1, \dots, o_n\}$ d’objets où chaque objet o_i est associé à une utilité u_i et un coût c_i . On supposera que ces paramètres ont des valeurs entières.
- *Output* : trouver un ensemble $S \subseteq O$ tel que $\sum_{s_i \in S} c_i \leq B$ (respect de la contrainte de coût) et qui maximise $u(S) = \sum_{s_i \in S} u_i$, la somme des utilités des objets sélectionnés.

La contrainte de coût représente traditionnellement le volume du sac à dos, mais elle peut aussi représenter une contrainte économique, ou un seuil réglementaire de pollution. Ce problème est difficile à résoudre ; il appartient à la classe des problèmes NP-difficiles. On se propose ici de regarder une variante de ce problème où il y a plusieurs contraintes de coût. Un objet o_i est donc associé à une utilité u_i et k valeurs de coût c_i^1, \dots, c_i^k . Il y a également k budgets B^1, \dots, B^k et le but est de trouver un ensemble $S \subseteq O$ tel que $\sum_{s_i \in S} c_i^l \leq B^l, \forall l \in \{1, \dots, k\}$ et qui maximise $u(S) = \sum_{s_i \in S} u_i$.

Pour le projet. Dans un package `sacADos`, vous fournirez des classes afin de représenter les concepts d’objet et d’instance du problème du sac-à-dos multidimensionnel. Un `Objet` possédera comme attributs une valeur **int** `utilite` et un tableau **int**[] `couts`. Un `SacADos` possédera notamment comme attributs une valeur **int** `dimension`, un tableau **int**[] `budgets` et une liste `List<Objet>` `objets`. Vous êtes libres de compléter cette description en rajoutant les méthodes, attributs et champs statiques que vous désirez ainsi que d’autres classes. Vous respecterez les bonnes pratiques d’encapsulation.

Résolution par méthodes gloutonnes

Pour trouver une bonne solution au problème du sac-à-dos multidimensionnel, nous allons commencer par des méthodes heuristiques simples, des méthodes gloutonnes. On distinguera les méthodes gloutonnes “à ajout” des méthodes gloutonnes “à retrait”.

Méthodes gloutonnes “à ajout”. Une méthode gloutonne “à ajout” construit un ensemble S d’objets de la manière suivante : partant de $S = \emptyset$, la méthode parcourt les objets du “plus intéressant” au “moins intéressant” (l’ordre “plus intéressant que” est défini selon un critère formel, cf paragraphe suivant) ; si l’objet peut être rajouté à S sans violer les contraintes de budget alors on l’ajoute, sinon non ; on passe ensuite à l’objet suivant et la méthode se termine quand on a inspecté tous les objets.

Plusieurs choix sont possibles pour définir la relation “plus intéressant que”. Nous proposons ici deux méthodes :

- soit $f_{\Sigma}(o_i) = u_i / (\sum_{l=1}^k c_i^l)$, un objet o_i est plus intéressant qu’un objet o_j si $f_{\Sigma}(o_i) \geq f_{\Sigma}(o_j)$.
- soit $f_{\max}(o_i) = u_i / (\max\{c_i^l, 1 \leq l \leq k\})$, un objet o_i est plus intéressant qu’un objet o_j si $f_{\max}(o_i) \geq f_{\max}(o_j)$.

Dans les deux cas, les égalités peuvent être cassées par un critère de votre choix.

Méthodes gloutonnes “à retrait”. Une méthode gloutonne “à retrait” construit un ensemble S d’objets de la manière suivante : partant de $S = O$, la méthode parcourt les objets du “moins intéressant” au “plus intéressant” ; si l’ensemble S actuel ne respecte pas les contraintes de budget, alors on enlève l’objet considéré et l’on passe au suivant ; si l’ensemble S respecte les contraintes de budget, alors on repart de l’objet courant avec une méthode gloutonne à ajout pour tenter de rajouter à nouveaux des objets. La méthode se termine à la fin de cette phase d’ajouts. Noter que le critère “plus intéressant que” peut ne pas être le même lors de la phase de retrait et celle d’ajout. Nous rajoutons un tel critère utile pour la phase de retrait :

- soit $f_{mv}(o_i) = u_i / \max(\{c_i^l, l \in L\})$ avec $L = \arg \max_{1 \leq l \leq k} (\sum_{o_i \in S} c_i^l - b^l)$, i.e., l’ensemble des dimensions avec le plus gros dépassement de budget. Un objet o_i est plus intéressant qu’un objet o_j si $f_{mv}(o_i) \geq f_{mv}(o_j)$.

Les égalités peuvent être cassées par un critère de votre choix.

Pour le projet. Dans un package `solveur.glouton`, définir des classes `GloutonAjoutSolver` et `GloutonRetraitSolver`. Ces classes définiront les méthodes gloutonnes vues plus haut. Ces méthodes prendront en argument un ou des paramètres `Comparator<Objet>` afin de comparer les objets selon différentes relations d’ordre. Vous êtes libres d’optimiser ces méthodes ou de proposer d’autres heuristiques ou critères d’ordre. Il devrait être facile de changer de méthode de résolution ou les critères d’ordre utilisés.

Résolution par méthode de Hill Climbing

Le *Hill Climbing* est une méthode simple de **recherche locale** permettant de trouver une bonne solution à un problème d’optimisation, ou à améliorer une solution d’un problème d’optimisation.

À partir d’une solution initiale (e.g., le résultat d’une de vos méthodes gloutonnes), la méthode du *Hill Climbing* explore les **voisins** (des solutions “proches”) de cette solution afin de trouver une meilleure solution. Si un voisin est effectivement meilleur (selon la fonction objectif), alors elle devient la nouvelle solution courante et la méthode continue. Sinon, la solution courante est un **optimum local** et la méthode se termine.

Voici un pseudo-code de la méthode du Hill Climbing.

Algorithm 1 fonction HILL-CLIMBING(*problem*)

retourne un optimum local.

problem est un problème d'optimisation pour lequel on peut définir une notion de voisinage entre les solutions admissibles et pour lequel f est la fonction objectif à maximiser.

```

1: solution ← solution initiale
2: while true do
3:   voisin ← un voisin qui maximise  $f$ 
4:   if  $f(\textit{voisin}) \leq f(\textit{solution})$  then
5:     return solution
6:   solution ← voisin
```

Pour le problème du sac à dos multidimensionnel, on peut définir la notion de voisinage de différentes façons. On vous propose la manière suivante : deux solutions admissibles (qui respectent les contraintes de budget) $S, S' \subseteq O$ sont voisines si $S' = (S \setminus E) \cup A$ avec $0 \leq |E| \leq t$, et $0 \leq |A| \leq t$, où t est une petite valeur que vous choisirez (e.g., $t = 1$ ou $t = 2$). Par exemple, si $t = 1$, alors S et S' sont voisins si on peut obtenir S' à partir de S en enlevant au maximum un objet, puis en rajoutant au maximum un objet.

L'approche du Hill Climbing a plusieurs limites :

- Cette approche fonctionne de manière myopique, elle se déplace dans l'espace des solutions admissibles en choisissant les déplacements un à un sans planification. Elle peut donc facilement rester coincée au niveau d'un maximum local ou d'un plateau. Pour permettre de dépasser ces plateaux, une variante permet un certain nombre de fois de se déplacer vers *voisin* même si $f(\textit{voisin}) = f(\textit{solution})$. Une autre variante consiste juste à utiliser cette méthode plusieurs fois à partir de solutions initiales différentes et ne garde que le meilleur résultat obtenu.
- Une autre limite du Hill Climbing pour le problème du sac à dos multidimensionnel est qu'il peut être trop coûteux d'explorer l'ensemble des voisins d'une solution. Une variante consiste alors à générer un nombre limité de voisins aléatoirement au lieu d'explorer tout le voisinage.

Pour le projet. Vous implémenterez les différentes classes nécessaires afin de pouvoir résoudre le problème du sac-à-dos multidimensionnel par la méthode du Hill Climbing. Vous pourrez explorer différentes variantes de cette méthode. Vous aurez une réflexion sur la modularité de votre approche.

Génération d'instances et lien avec notre équipe municipale

Vous fournirez une classe `VersSacADos` avec des méthodes permettant de générer un objet de la classe `SacADos` à partir d'une liste de projets (c.f. section sur notre équipe municipale) et des budgets alloués par Dauphine City. Nous proposons deux options différentes :

- Les budgets concernent les 3 différents types de coûts (économique, social, et environnemental).
- Les budgets concernent les 5 différents secteurs et seul le coût économique est considéré. Dit autrement, il y a un budget de B^1 euros pour le sport, de B^2 euros pour la santé, . . .

Afin de tester vos méthodes sur des instances plus difficiles vous fournirez également dans la même classe un méthode permettant d'obtenir une instance de la classe `SacADos` à partir de certains fichiers téléchargeables ici ¹.

1. Drake, John. (2015). Benchmark instances for the Multidimensional Knapsack Problem. 10.13140/2.1.3578.9122.

Chacun de ces fichiers obéit au format suivant :

- n le nombre d’objets, k le nombre de budgets, valeur optimale de la solution, 0 si inconnue.
- Utilité u_i pour chacun des n objets.
- Matrice de contraintes de taille $(k \times n)$, i.e., on trouve c_i^l à l’indice $[l, i]$.
- Budget B^l pour $l \in \{1, \dots, k\}$.

La partie Main

Votre projet contiendra une classe `Main` dont la méthode `main` permettra de tester votre programme dans plusieurs cas de figure (e.g., les contraintes de coût correspondent au différents types d’impact ou aux différents secteurs).

Dans chaque cas de figure, des interactions auront lieu dans la console, à l’aide d’un menu déroulant interactif et robuste.

Quelques consignes et points d’attention

- Votre projet possédera une documentation Javadoc de qualité.
 - Votre projet sera structuré en utilisant des packages.
 - Votre projet respectera les bonnes pratiques d’encapsulation. Notamment, vous utiliserez à bon escient les notions de classes abstraites et d’interfaces afin de bien séparer les notions de contrat et d’implémentation.
 - Votre projet utilisera des annotations quand cela est pertinent.
 - Vous aurez une réflexion sur les différentes structures de données exploitées lors de l’utilisation de collections et de maps.
 - Votre code gèrera de manière adaptée les différentes exceptions possibles pouvant être levées lors de l’exécution du programme.
 - Vous choisirez 5 méthodes de votre choix sur lesquels vous effectuerez des tests JUnit. Une attention particulière sera portée sur la qualité de vos tests.
 - Votre code doit pouvoir être testé facilement et de manière intuitive.
- Utilisation de fichiers pour évaluer partie sac à dos multidimensionnel.

Ce travail est à réaliser **en trinôme**. En plus de votre code, vous devrez rendre un *rapport* contenant :

- Une répartition de votre travail entre vous.
- Une présentation de la structure de votre code, et une justification des choix concernant vos hiérarchies de classes.
- Une discussion sur ce qui a été le plus dur à implémenter pour vous et sur le niveau de difficulté du projet.

→ Votre rapport contiendra également un retour critique et honnête sur votre utilisation des IAs génératives. Vous communiquerez 1) les différents types et le nombre de prompts utilisés, 2) les retours critiques que vous avez eus sur le code et le texte généré, et 3) les avantages et les inconvénients que vous avez identifiés sur l’utilisation de ces outils.

Votre projet devra également contenir un fichier *ReadMe* présentant succinctement votre projet et détaillant les instructions à réaliser pour le compiler, l’exécuter, puis l’utiliser.

Ce projet compte pour 40% de la note de l'UE de Java. Il est donc souhaitable que la note corresponde au travail de votre groupe, et non aux conseils d'autres groupes, d'autres étudiants ou d'internet. Si vous utilisez des sources (articles de recherche, posts sur internet, etc...), vous devez mentionner vos sources dans le rapport.